07 | WASI：你听说过 WebAssembly 操作系统接口吗?

2020-09-18 于航

WebAssembly入门课 进入课程 >

讲述：于航
时长 18:13 大小 16.69M

你好，我是于航。

相信你在刚刚接触到 WebAssembly 这门技术的时候一定有所发现，WebAssembly 这个单词实际上是由两部分组成，也就是 “Web” 和 “Assembly” 。

“Web” 表明了 Wasm 的出身，也就是说它发明并最早应用于 Web 浏览器中，“Assembly” 则表明了 Wasm 的本质，这个词翻译过来的意思是 “汇编” ，也就是指代它的 V-ISA 属性。

鉴于 Wasm 所拥有 “可移植” 、 “安全” 及 “高效” 等特性，Wasm 也被逐渐应用在 Web 领域之外的一些其他场景中。今天我们将要讲解的，便是可以用于将 Wasm 应用到

out-of-web 环境中的一项新的标准 —— WASI（WebAssembly System Interface，Wasm 操作系统接口）。通过这项标准，Wasm 将可以直接与操作系统打交道。

在正式讲解 WASI 之前，我们先来学习几个与它息息相关的重要概念。在了解了这些概念之后，相信甚至不用我过多介绍，你也能够感受到 WASI 是什么，以及它是如何与 Wasm 紧密结合的。

Capability-based Security

第一个我们要讲解的，是一个在“计算机安全”领域中十分重要的概念 —— “Capability-based Security”，翻译过来为“基于能力的安全”。由于业界没有一个相对惯用的中文表达方式，因此我还是保持了原有的英文表达来作为本节的标题，在后面的内容中，我也将直接使用它的英文表达方式，以保证内容的严谨性。

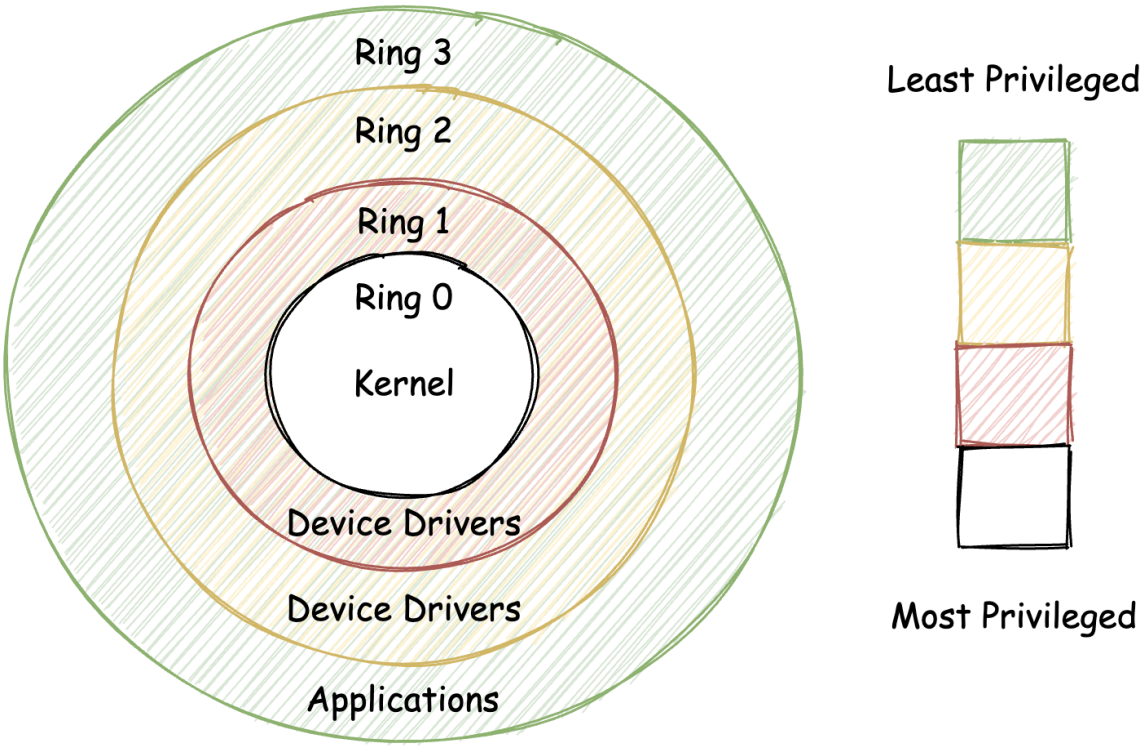
Capability-based Security 是一种已知的、常用的安全模型。通常来讲，在计算机领域中，我们所提及的 capability 可以指代如 Token、令牌等概念。capability 是一种用于表示某种权限的标记，它可以在用户之间进行传递且无法被伪造。

在一个使用了 Capability-based Security 安全模型的操作系统中，任何用户对计算机资源的访问，都需要通过一个具体的 capability 来进行。

Capability-based Security 同时也指代了一种规范用户程序的原则。比如这些用户程序可以根据“最小特权原则”（该原则要求计算环境中的各个模块仅能够访问当下所必需的信息或资源）来彼此直接共享 capability，这样可以使得操作系统仅分配用户程序需要使用的权限，并且可以做到“一次分配，多次使用”。

Capability-based Security 这个安全模型，通常会跟另外的一种基于“分级保护域”方式实现的安全模型形成对比。

基于“分级保护域”实现的安全模型，被广泛应用于类 Unix 的各类操作系统中，比如下图所示的操作系统 Ring0 层和 Ring3 层（Ring1 / Ring2 一般不会被使用）便是“分级保护域”的一种具体实现形式。



在传统意义上，Ring0 层拥有着最高权限，一般用于内核模式；而 Ring3 层的权限则会被稍加限制，一般用于运行用户程序。当一个运行在 Ring3 层的用户程序，试图去调用只有 Ring0 层进程才有权限使用的指令时，操作系统会阻止调用。这就是“分级保护域”的大致概念。

反观 Capability-based Security，capability 通过替换在分级保护域中使用的“引用”，来达到提升系统安全性的目的。这里的“引用”是指用于访问资源的一类“定位符”，比如用于访问某个文件资源的“文件路径字符串”便是一个引用。

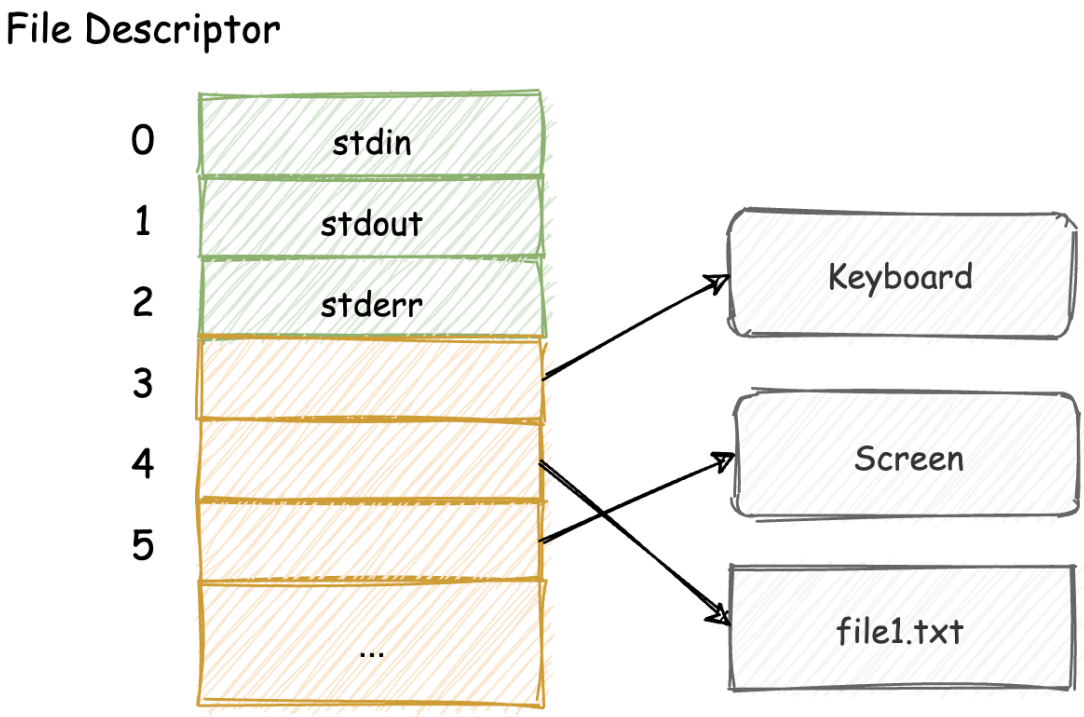
引用本身并没有指定实际对应资源的权限信息，以及哪些用户程序可以拥有这个引用。因此，每一次尝试通过该引用来访问实际资源的操作，都会经由操作系统来进行基于“分级保护域”的权限验证。比如验证发起访问的用户是否有权限持有该资源，这种方式便十分适合早期计算机系统的“多用户”特征（每个用户有不同的权限）。

在具有 capability 概念的操作系统中，只要用户程序拥有了这个 capability，那它就拥有足够的权限去访问对应的资源。从理论上来讲，基于 Capability-based Security 的操作系统，甚至不需要如“权限控制列表（ACL）”这类的传统权限控制机制。

当然，为了实现上述我们提到的 capability 的能力，每一个 capability 不再是单一的由“字符串”组成的简单数据结构。并且我们还需要保障，capability 的内部结构不会被

用户程序直接访问和修改，以防止 capability 本身被伪造。

相对应的，用户程序只能够通过 capability 暴露出的特定“入口”，来访问对应的系统资源。我们可以用操作系统中常见的一个概念 —— “文件描述符（File Descriptor）”来类比 capability 的概念。如下图所示。



我们可以将文件描述符类比为 capability。举个例子，当应用程序在通过 C 标准库中的“fopen” 函数去打开一个文件时，函数会返回一个非负整数，来表示一个特定文件资源对应的文件描述符。

在拥有了这个描述符后，应用程序便可以按照在调用 “fopen” 函数时所指定的操作（比如 “w”），来相应地对这个文件资源进行处理。当函数返回负整数时，则表示无法获得该资源。在这些返回的错误代码中，就包含有与“权限不足”相关的调用错误信息。

最为重要的一点是，拥有某个 capability 的用户程序，可以“任意地”处理这个 capability。比如，可以访问其对应的系统资源、可以将其传递给其他的应用程序来进行使用，或者也可以选择直接将这个 capability 删除。操作系统有义务确保某个特定的 capability 只能够对应系统中的某个特定的资源或操作，以保证安全策略的完备性。

系统调用（System Call）

第二个我们要讲解的概念叫做 “System Call”，翻译成中文即“系统调用”（或者也可称为 “操作系统调用”，这里我们使用简称 “系统调用”）。

还是回到我们在上一小节中曾提到过的一个场景：“使用 C 标准库中的 fopen 函数，来打开一个计算机本地文件”。请试想，当我们在调用这个 fopen 函数打开某个文件时，实际上发生了什么？fopen 函数是如何访问操作系统的文件资源的呢？带着这两个问题，我们一步步来看。

既然我们说 fopen 函数是 C 标准库中定义的一个函数，那么我们就从某个特定的 C 标准库实现所对应的源代码入手，来看看 fopen 函数的具体实现细节。这里我们以 musl 这个 libc 的实现为例。在它的源代码中，我们可以找到如下这段对 fopen 函数的定义代码（这里只列出了关键的部分）。

复制代码

```
FILE *fopen(const char *restrict filename, const char *restrict mode) {
  ...
  /* Compute the flags to pass to open() */
  flags = __fmodeflags(mode);

  fd = sys_open(filename, flags, 0666);
  if (fd < 0) return 0;
  ...
}
```

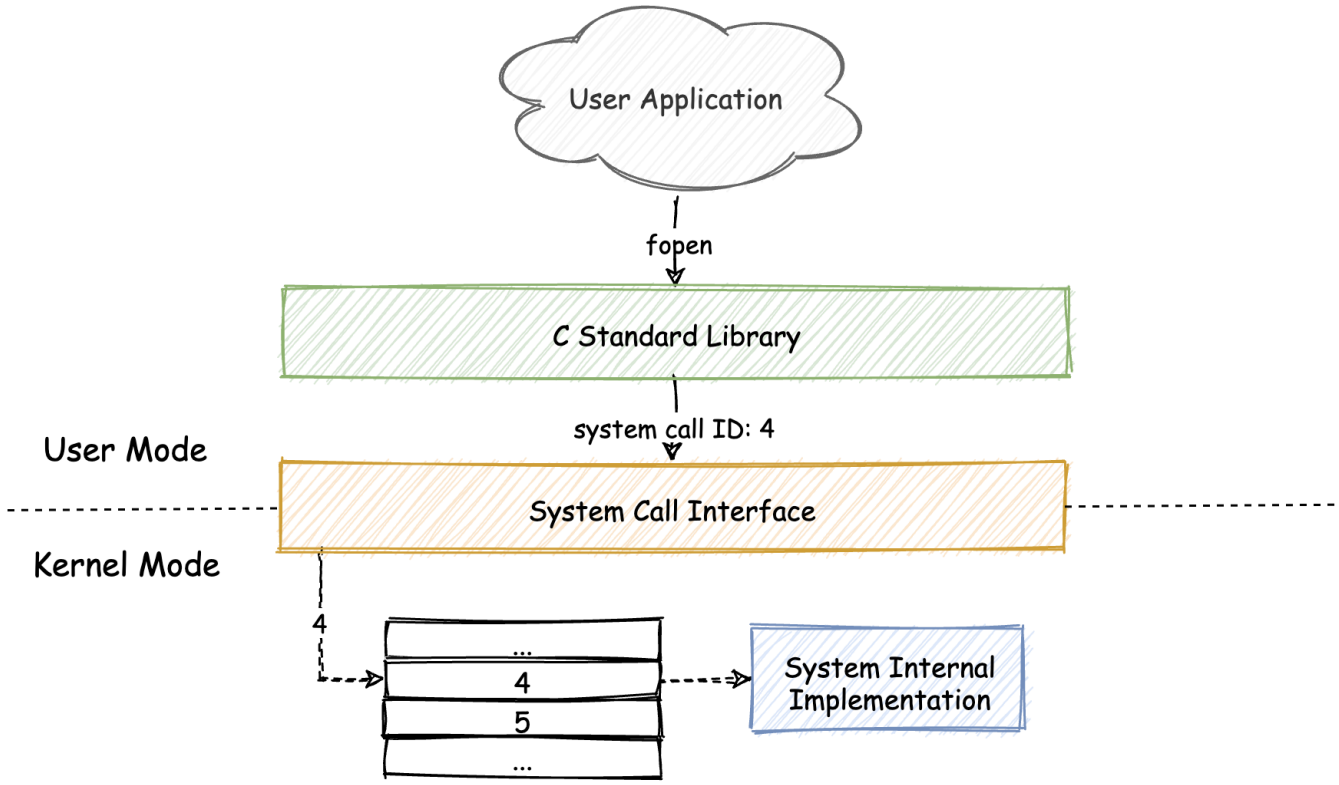
代码的具体实现流程和细节我们不做深究，唯一你需要注意的，就是这句函数调用语句 “*fd = sys_open(filename, flags, 0666);*”。在这行语句中，musl 调用了一个名为 “**sys_open**” 的函数，而在这个函数的背后，就是我们本节内容的主角 —— “系统调用”。

实际上，任何其他需要与操作系统资源打交道的 C，甚至是 C++ 标准库函数（包括 fopen 函数在内），都需要通过 “系统调用” 来间接访问和使用这些系统资源。sys_open 函数其实是对系统调用进行了封装，在函数内部会使用内联的汇编代码，去实际调用某个具体的“系统调用”。这里 sys_open 对应的，便是指“用于打开本地文件资源”的那个系统调用。

每一个系统调用，都对应着需要与操作系统打交道的某个特定功能，并且有着唯一的“系统调用 ID” 与之相对应。在不同的操作系统中，对应同一系统调用的系统调用 ID 可能会发生变化。

而 C/C++ 标准库的作用，便是为我们提供了一个统一、稳定的编程接口。让我们的程序可以做到“**一次编写，到处编译**”。从某种程度上来讲，标准库的出现为应用程序源代码提供了“可移植性”。比如让我们不再需要随着操作系统类型的变化，而硬编码不同的系统调用 ID。

除此之外，标准库还会帮助我们处理系统调用前后需要做的一些事情，比如简化函数参数的传递、对各种异常情况进行处理，以及“关闭文件”之类的“善后”工作。关于用户应用程序与操作系统调用之间的关系，你可以参考我下面绘制的这幅图。



WebAssembly 操作系统接口（WASI）

好了，在讲解完 “Capability-based Security” 以及“系统调用”这两个概念之后，我们再把目光移回到今天的主角 —— WASI。其实从 WASI 对应的全称中，我想你能够猜测到，它肯定与我们在上一节中介绍的“系统调用”有着某种联系（System Call 与 System Interface）。没错，那么接下来我们就一起看看 WASI 究竟是什么。

我们从“如何在 Web 场景之外使用 Wasm？”这个问题开始说起。我们都知道，Wasm 是一套新的 V-ISA（也就是“虚拟指令集架构”），其中的这些虚拟指令便无法被真实的物理 CPU 硬件直接执行。

所以如果我们想要在浏览器之外使用 Wasm，就需要提供一种基础设施，来解释并执行这些被存放在 Wasm 二进制模块中的虚拟指令。对于这样的基础设施，我们通常称之为“虚拟机（Virtual Machine）”，或者是 “运行时引擎（Runtime Engine）”。

OK，假设此时我们已经有了这样的一个虚拟机，可以用于执行 Wasm 的虚拟字节码指令。然后我们希望将这样一段 C/C++ 代码经过编译后，以 Wasm 的形式在这个虚拟机中运行。在这段 C/C++ 代码中，我们使用到了之前提到的 fopen 函数。

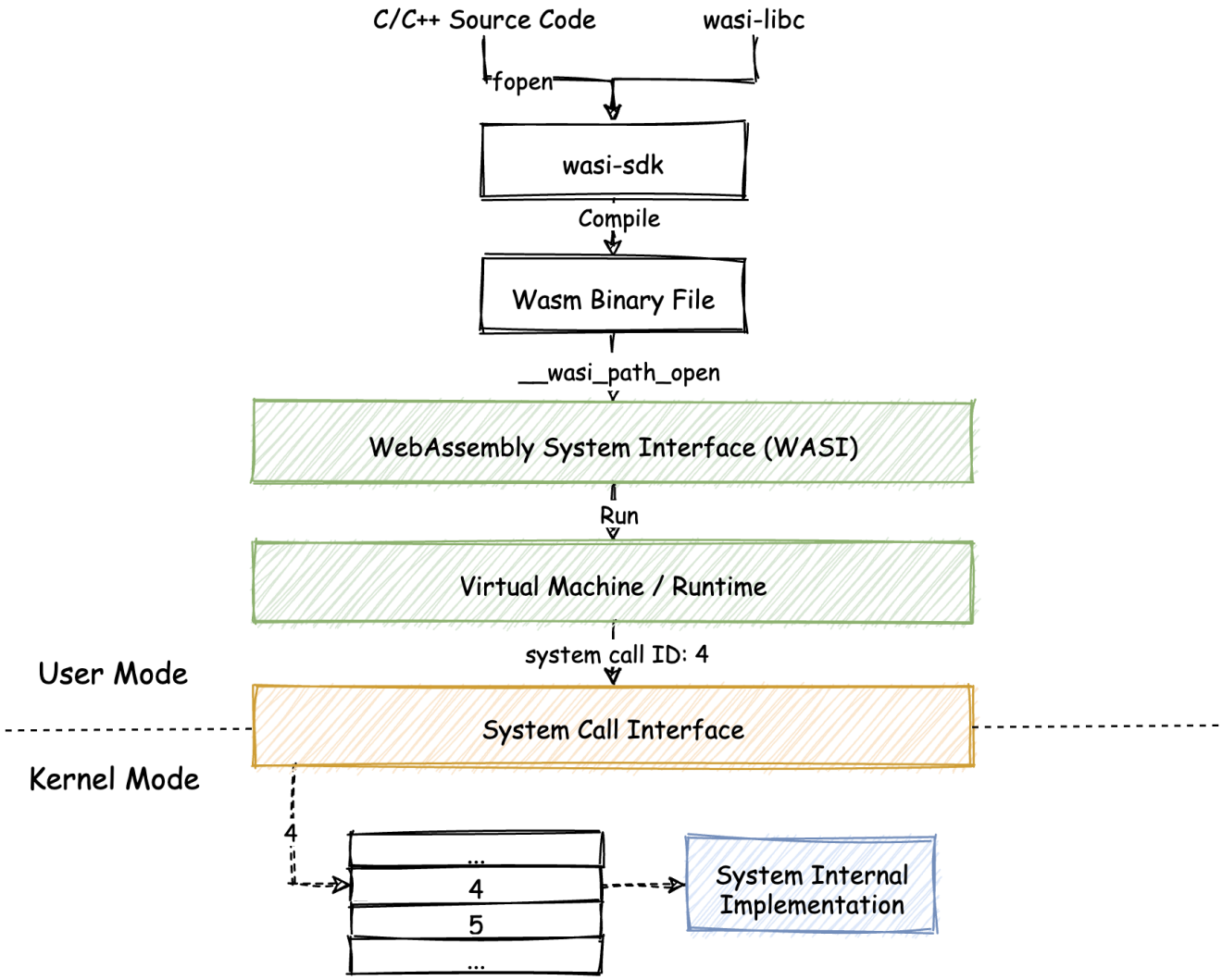
但是问题来了。我们知道，在如 musl 这类 C 标准库的实现中，类似 fopen 这样的函数，最后会被编译为对某个特定平台（IA32、X86-64 等）系统调用的调用过程。这对于 Wasm 来说，会使自己丧失“天生自带”的可移植性。

单纯对于某一个 Wasm 模块来讲，由于我们并不知道这个模块将会被运行在什么类型的操作系统上，因此我们无法将平台相关的具体信息放到 Wasm 模块中。那如何解决这个问题呢？WASI 给了我们答案。

WASI 在 Wasm 字节码与虚拟机之间，增加了一层“系统调用抽象层”。比如对于在 C/C++ 源码中使用的 fopen 函数，当我们将这部分源代码与专为 WASI 实现的 C 标准库 “wasi-libc” 进行编译时，源码中对 fopen 的函数调用过程，其内部会间接通过调用名为 “__wasi_path_open” 的函数来实现。这个 __wasi_path_open 函数，便是对实际系统调用的一个抽象。

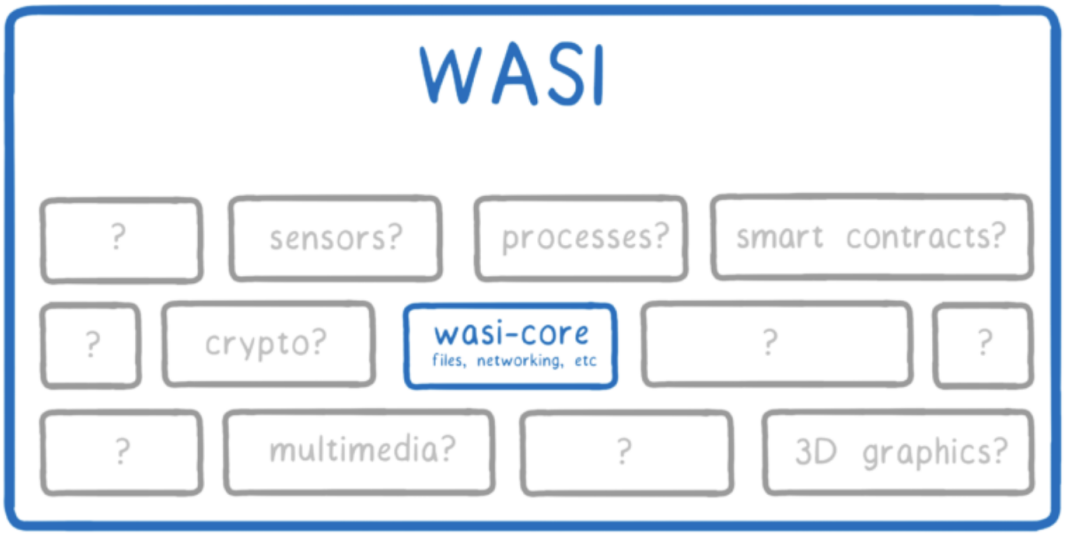
__wasi_path_open 函数的具体实现细节会交由各个虚拟机自行处理。也就是说，虚拟机需要在其 Runtime 运行时环境中提供，对 Wasm 模块字节码所使用到的 __wasi_path_open 函数的解析和执行能力的支持。而虚拟机在实际实现这些系统调用抽象层接口时，也需要通过实际的系统调用来进行。只不过这些细节上的处理，对于 Wasm 二进制模块来讲，是完全透明的。

我们可以将上述提到的 wasi-libc、Wasm 二进制模块、WASI 系统调用抽象层，以及虚拟机基础设施之间的关系，通过下图来直观地展示。



实际上，类似 __wasi_path_open 的这类以 “__wasi” 开头的，用于抽象实际系统调用的函数，便是 WASI 的核心组成部分。WASI 根据不同系统调用所提供的不同功能，将这些系统调用对应的 WASI 抽象函数接口，分别划分到了不同的子集合中。

如下图所示，一个名为 “wasi-core” 的 WASI 标准子集合，包含有对应于“文件操作”与“网络操作”等相关系统调用的 WASI 抽象函数接口。其他如 “crypto”、“multimedia” 等子集合，甚至可以包含与实际系统调用无关的一系列 WASI 抽象系统调用接口。你可以理解为 WASI 所描述的抽象系统调用，是针对 Wasm V-ISA 描述的抽象机器而言。针对这部分抽象系统的具体实现，则会依赖一部分实际的系统调用。



图片来源于 Lin Clark 博客

WASI 在设计和实现时，需要遵守 Wasm 的“可移植性”及“安全性”这两个基本原则。那下面我们来分别看一看， WASI 及其相关的运行时 / 虚拟机基础设施，是如何确保能够在设计和实现时满足这两个基本原则的。

可移植性（Portability）

对于“可移植性”，其实我们已经在讲解 WASI 时给出了答案。WASI 通过在 Wasm 二进制字节码与虚拟机基础设施之间，提供统一的“系统调用抽象层”来保证 Wasm 模块的可移植性。这样一来，上层的 Wasm 模块可以不用考虑平台相关的调用细节，统一将对实际系统调用的调用过程，转换为对“抽象系统调用”的调用过程。

而“抽象系统调用”的实现细节，则由下层的相关基础设施来负责处理。基础设施会根据其所在操作系统类型的不同，将对应的抽象系统调用映射到真实的系统调用上。当然，并不是所有的抽象系统调用都需要被映射到真实的系统调用上，因为对于某些抽象系统调用而言，基础设施只是负责提供相应的实现即可。

这样，一个经过编译生成的 Wasm 二进制模块便可以在浏览器之外也同样保证其可移植性。真正做到“一次编译，到处运行”， “**抽象**”便是解决这个问题的关键。

安全性（Security）

对于“安全性”，我们需要再次回到开头介绍的 “Capability-based Security”。

实际上，基础设施在真正实现 WASI 标准时，便会采用 “Capability-based Security” 的方式来控制每一个 Wasm 模块实例所拥有的 capability。

举个例子，假设一个 Wasm 模块想要打开一个计算机本地文件，而且这个模块还是由使用了 fopen 函数的 C/C++ 源代码编译而来，那对应的虚拟机在实例化该 Wasm 模块时，便会将 fopen 对应的 WASI 系统调用抽象函数 “__wasi_path_open” 以某种方式（比如通过包装后的函数指针），当做一个 capability 从模块的 Import Section 传递给该模块进行使用。

通过这种方式，基础设施掌握了主动权。它可以决定是否要将某个 capability 提供给 Wasm 模块进行使用。若某个 Wasm 模块偷偷使用了一些不为开发者知情的系统调用，那么当该模块在虚拟机中进行实例化时，便会露出马脚。掌握这样的主动权，正适合如今我们基于众多不知来源的第三方库进行代码开发的现状。

对于没有经过基础设施授权的 capability 调用过程，将会被基础设施拦截。通过相应的日志系统进行收集，这些“隐藏的小伎俩”便会在第一时间被开发者 / 用户感知，并进行相应的处理。

总结

好了，讲到这，今天的内容也就基本结束了。最后我来给你总结一下。

本节课，我们主要讲解了什么是 WASI。WASI 通过增加“抽象层”的方式，解决了 Wasm 抽象机器（V-ISA）与实际操作系统调用之间的可移植性问题，这可以保证我们基于 WASI 编写的 Wasm 应用（模块）真正做到“**一次编译，到处运行**”。抽象出的“Wasm 系统调用层”将交由具体的底层基础设施（虚拟机 / 运行时）来提供实现和支持。

不仅如此，基于 Capability-based Security 模型，WASI 得以在最大程度上保证 Wasm 模块的运行时安全。通过配合 Wasm 模块的 Import Section 与 Export Section，运行时便可以细粒度地控制模块实例所能够使用的系统资源，这相较于传统的“分级保护域”模型来说，无疑会更加灵活和安全。每一个 Wasm 模块在运行时都仅能够使用被授权的 capability，而 WASI 中定义的这些系统调用抽象接口便属于众多 capability 中的一种。

另外你还需要知道的一点是，无论是 Capability-based Security 模型，还是“分级保护域”模型，两者都是如今被广泛使用的安全模型。只不过相对来说，“最小特权原则”使得 Capability-based Security 模型对权限的控制力度会更加精细，而“分级保护域”模型则是操作系统中广泛使用的一种安全策略。

课后思考

最后，我们来做一个思考题吧。

你还能举出哪些场景，是通过增加“抽象层”来解决了某个实际问题的？

今天的课程就结束了，希望可以帮助到你，也希望你在下方的留言区和我参与讨论，同时欢迎你把这节课分享给你的朋友或者同事，一起交流一下。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 06 | WAT：如何让一个 WebAssembly 二进制模块的内容易于解读?

下一篇 08 | API：在 WebAssembly MVP 标准下你能做到哪些事?

精选留言 (4)

写留言

popgis
2020-09-18

请问老师，那WASI是将系统调用函数在编译期替换成WASI标准函数的吗？还是在运行期通过调用拦截或动态替换的方式?

展开

作者回复: 对的，应用在与wasi-libc编译时便会被替换成WASI的系统调用。然后虚拟机在解释执行Wasm字节码时就会使用虚拟机上的实现。

1 2

Cryhard
2020-09-18

那这意味着Wasm的编写也需要像原生APP一样，运行中申请一组由WASI“代理”的系统权限的集合（而用户可以提供长期或者单次授权）。而程序也得支持“部分权限被禁用”的情况，例如在“不能调用本地录音设备”的状态下继续正常运行吗?

展开

作者回复: 是的，这个是一种情况。但实际上对 WASI 接口的调用需要进行怎样的权限管理要依据不同的情况来看。比如最简单的用虚拟机执行一个 WASI 应用，一般可能只要在运行虚拟机时指定所需要的“权限”接可。还有一种场景就是在云上使用 WASI。这个时候可以把 WASI 应用比作 Actor，每个 Actor 都需要显式指出自己需要使用的系统权限（WASI 接口）。如果使用了没有声明的注册的权限，一般会被直接注销或者shutdown。因为这里的 Actor 更像是注册一个 service。原生应用就又是另外一种形式了。

1

军秋
2020-09-26

除了浏览器和node，还有其它可运行WASI的运行环境吗？或者如何自己构建这个运行环境？

作者回复: 有的，有很多开源的优秀 Wasm 运行时可以使用，它们大部分也都支持 WASI。这个我会在后面的文章中介绍哈。

一步

2020-09-20

wasm 可以直接运行在 Chrome 浏览器上 V8 引擎的，不存在系统调用。 那么可以直接使用的 node 环境吗？（node 的引擎也是 v8, 但是有系统调用） 还是要需要 引入 wasi 来重新编译 node 才能使用的？

作者回复: Node 好像已经部分支持 WASI 了，可以看看这里：http://nodejs.cn/api/wasi.html